Sorting Room

Welcome to Sorting Room on Exercism's Go Track. If you need help running the tests or submitting your code, check out `HELP.md`. If you get stuck on the exercise, check out `HINTS.md`, but try and solve it without using those first :)

Introduction

Type Conversion

Go requires explicit conversion between different types. Converting between types (also known as **type casting**) is done via a function with the name of the type to convert to. For example, to convert an `int` to a `float64` you would need to do the following:

```
var x int = 42 // x has type int
f := float64(x) // f has type float64 (ie. 42.0)
```

Converting between primitive types and strings

There is a `strconv` package for converting between primitive types (like `int`) and `string`.

```
import "strconv"

var intString string = "42"
var i, err = strconv.Atoi(intString)

var number int = 12
var s string = strconv.Itoa(number)
```

Type Assertions

Interfaces in Go can introduce ambiguity about the underlying type. A type assertion allows us to extract the interface value's underlying concrete value using this syntax: `interfaceVariable.(concreteType)` .

For example:

```
var input interface{} = 12
number := input.(int)
```

NOTE: this will cause a panic if the interface variable does not hold a value of the concrete type.

We can test whether an interface value holds a specific concrete type by making use of both return values of the type assertion: the underlying value and a boolean value that reports whether the assertion succeeded. For example:

```
str, ok := input.(string) // no panic if input is not a string
```

If `input` holds a `string` , then `str` will be the underlying value and `ok` will be true. If `input` does not hold a `string` , then `str` will be the zero value of type `string` (ie. `""` - the empty string) and `ok` will be false. No panic occurs in any case.

Type Switches

A **type switch** can perform several type assertions in series. It has the same syntax as a type assertion (`interfaceVariable.(concreteType)`), but the `concreteType` is replaced with the keyword `type` . Here is an example:

```
var i interface{} = 12 // try: 12.3, true, int64(12), []int{}, map[string]int{}

switch v := i.(type) {
case int:
    fmt.Printf("the integer %d\n", v)
case string:
    fmt.Printf("the string %s\n", v)
default:
    fmt.Printf("type, %T, not handled explicitly: %#v\n", v, v)
}
```

Instructions

Jen is working in the sorting room in a large factory. The sorting room needs to process anything that comes into it by categorizing it with a label. Jen is responsible for things that were pre-categorized as numbers and needs a program to help her with the sorting.

Most primitive values should get straight-forward labels. For numbers, she wants strings saying `"This is the number 2.0"` (if the number was 2). Jen wants the same output for integers and floats.

There are a few `Box` interfaces that need to be unwrapped to get their contents. For a `NumberBox`, she wants strings saying `"This is a box containing the number 3.0"` (if the `Number()` method returns 3). For a `FancyNumberBox`, she wants strings saying `"This is a fancy box containing the number 4.0"`, but only if the type is a `FancyNumber`.

Anything unexpected should say `"Return to sender"` so Jen can send them back where they came from.

1. Describe simple numbers

Jen wants numbers to return strings like `"This is the number 2.0"` (including one digit after the decimal):

```
fmt.Println(DescribeNumber(-12.345))
// Output: This is the number -12.3
```

2. Describe a number box

Jen wants number boxes to return strings like `"This is a box containing the number 2.0"` (again, including one digit after the decimal):

```
fmt.Println(DescribeNumberBox(numberBoxContaining{12}))
// Output: This is a box containing the number 12.0
```

3. Implement a method extracting the number from a fancy number box

Jen needs a helper function to extract the number from a `FancyNumberBox`. If the `FancyNumberBox` is a `FancyNumber`, extract its value and convert it from a `string` to an `int`. Any other type of `FancyNumberBox` should return 0.

```
fmt.Println(ExtractFancyNumber(FancyNumber{"10"}))
// Output: 10
fmt.Println(ExtractFancyNumber(AnotherFancyNumber{"4"}))
// Output: 0
```

4. Describe a fancy number box

If the `FancyNumberBox` is a `FancyNumber`, Jen wants strings saying `"This is a fancy box containing the number 4.0"`. Any other type of `FancyNumberBox` should say `"This is a fancy box containing the number 0.0"`.

```
fmt.Println(DescribeFancyNumberBox(FancyNumber{"10"}))
// Output: This is a fancy box containing the number 10.0
fmt.Println(DescribeFancyNumberBox(AnotherFancyNumber{"4"}))
// Output: This is a fancy box containing the number 0.0
```

NOTE: we should use the `ExtractFancyNumber` function!

5. Implement `DescribeAnything` which uses them all

This is the main function Jen needs which takes any input (the empty interface means any value at all: `interface{}`). `DescribeAnything` should delegate to the other functions based on the type of the value passed in. More specifically:

- `int` and `float64` should both delegate to `DescribeNumber`
- `NumberBox` should delegate to `DescribeNumberBox`
- `FancyNumberBox` should delegate to `DescribeFancyNumberBox`
- anything else should result in `"Return to sender"`

```
fmt.Println(DescribeAnything(numberBoxContaining{12.345}))
// Output: This is a box containing the number 12.3
fmt.Println(DescribeAnything("some string"))
// Output: Return to sender
```

Source

Created by

- @jmrunkle

Contributed to by

- @junedev